Algorand Blockchain
Introduction to blockchain and smart contract design
Part II: Algorand Transactions and dApps

Giuseppe Persiano

Università di Salerno
giuper@gmail.com

A Python Approach

Objective

Design and implementation of a *Distributed Autonomous Organization (DAO)* on the blockchain.

Our DAO

- it holds a certain number of tokens `FOSAD22Token`
- anyone can buy the tokens at the current price
- governors can set the price
- governors can sell their right
- when all tokens are sold, DAO dissolves and shares the proceeds among its founders

Distributed: once setup, it runs on the distributed blockchain

Autonomous: no human supervision is needed once it is started

Centralized

One program running on one machine on the Internet

Trust assumption: One program is trusted

Decentralized

One program run by the whole network

Trust assumption: Two thirds of the network are trusted

How we leverage on widespread trustfulness

Algorand Transactions

There are six types of transactions in Algorand

1. Payment: to send Algo from one account to another

Source: https://developer.algorand.org/docs/get-details/transactions/

Algorand Transactions

There are six types of transactions in Algorand

1. Payment: to send Algo from one account to another
2. Key Registration: to register a key to take part in consensus

Source: https://developer.algorand.org/docs/get-details/transactions/

Algorand Transactions

There are six types of transactions in Algorand

1. Payment: to send Algo from one account to another
2. Key Registration: to register a key to take part in consensus
3. Asset Configuration: to configure an *asset*

Source: https://developer.algorand.org/docs/get-details/transactions/

Algorand Transactions

There are six types of transactions in Algorand

1. Payment: to send Algo from one account to another
2. Key Registration: to register a key to take part in consensus
3. Asset Configuration: to configure an *asset*
4. Asset Freeze: to freeze an *asset*

Source: https://developer.algorand.org/docs/get-details/transactions/

Algorand Transactions

There are six types of transactions in Algorand

1. Payment: to send Algo from one account to another
2. Key Registration: to register a key to take part in consensus
3. Asset Configuration: to configure an *asset*
4. Asset Freeze: to freeze an *asset*
5. Asset Transfer: to transfer *asset*

Source: https://developer.algorand.org/docs/get-details/transactions/

Algorand Transactions

There are six types of transactions in Algorand

1. Payment: to send Algo from one account to another
2. Key Registration: to register a key to take part in consensus
3. Asset Configuration: to configure an *asset*
4. Asset Freeze: to freeze an *asset*
5. Asset Transfer: to transfer *asset*
6. Application Calls: to call an *application*

Source: https://developer.algorand.org/docs/get-details/transactions/

Payment Transactions


```json
{
  "txn": {
    "amt": 5000000,
    "fee": 1000,
    "fv": 6000000,
    "gen": "mainnet-v1.0",
    "gh": "wGHE2Pwdvd7S12BL5FaOP20EGYesN73ktiC1qzkkit8=",
    "lv": 6001000,
    "note": "SGVsbG8gV29ybGQ=",
    "rcv": "GD64YIY3TWGDMCNPP553DZPPR6LDUSFQOIJVFDPPXWEG3FVOJCCDBBHU5A",
    "snd": "EW64GC6F24M7NDSC5R3ES4YUVE3ZXXNMARJHDCCCLIHZU6TBEOC7XRSBG4",
    "type": "pay"
  }
}
```

Payment Transactions: Fields

- `type`: `pay`
- `amt`: the total amount to be sent in microAlgos.
- `fee`: paid by the sender to the `FeeSink` to prevent denial-of-service. Minimum fee: 1000 microAlgos.
- `snd`: the address of the account that pays the fee and amount.
- `rcv`: the address of the account that receives the amount.
- `fv`, `lv`: the first and last round for when the transaction is valid. If the transaction is sent prior to `fv` or after `lv`, it will be rejected by the network.
- `gen`, `gh`: the genesis block name and hash.

Source: https://developer.algorand.org/docs/get-details/transactions/transactions/

Closing an account

```json
{
  "txn": {
    "close": "EW64GC6F24M7NDSC5R3ES4YUVE3ZXXNMARJHDCCCLIHZU6TBEOC7XRSBG4",
    "fee": 1000,
    "fv": 4695599,
    "gen": "testnet-v1.0",
    "gh": "SGO1GKSzyE7IEPItTxCByw9x8FmnrCDexi9/cOUJOiI=",
    "lv": 4696599,
    "rcv": "EW64GC6F24M7NDSC5R3ES4YUVE3ZXXNMARJHDCCCLIHZU6TBEOC7XRSBG4",
    "snd": "SYGHTA2DR5DYFWJE6D4T34P4AWGCG7JTNMY4VI6EDUVRMX7NG4KTA2WMDA",
    "type": "pay"
  }
}
```

Pay `amt` to `snd` and the remaining balance of `rcv` is transferred to `close`.

Constructing a payment TX using the python SDK

Step 1

- Create the transaction

```
unsignedTx=PaymentTxn(sAddr,params,rAddr,amount,None,note)
```



```
{
  "txn": {
    "amt": 1000000,
    "fee": 1000,
    "fv": 17274399,
    "gen": "testnet-v1.0",
    "gh": "SGO1GKSzyE7IEPItTxCByw9x8FmnrCDexi9/cOUJOiI=",
    "lv": 17275399,
    "note": "Q2lhbyBQaW5vISEh",
    "rcv": "CHCJJOOLATSUEEILZF5NGAGFS3JCUXG4I6EWTMTWVRLTJDGA6DKG5NKPA4",
    "snd": "DF2QZX26LUCBOGLQWAXJT56OJFHSZ2V4UFD3SEXXASEVZAIZV6FZSJ2HPM",
    "type": "pay"
  }
}
```

Signing a transaction

A transaction must be signed.

1. single-account signature
2. multi-account signature
3. smart signature

Constructing a payment TX using the python SDK

Step 2

- Sign the transaction

```
signedTx=unsignedTx.sign(sKey)
```


```
{
  "sig": "6GCon/O1EDTJUvZMJL+NydF+9Spl9kDoJ5UXJhQKHZnwFsBeSYyZ/NSZDdAkcvSeqKv0phJch9mL/Yp16/9tAw==",
  "txn": {
    "amt": 1000000,
    "fee": 1000,
    "fv": 17274399,
    "gen": "testnet-v1.0",
    "gh": "SGO1GKSzyE7IEPItTxCByw9x8FmnrCDexi9/cOUJOiI=",
    "lv": 17275399,
    "note": "Q2lhbyBQaW5vISEh",
    "rcv": "CHCJJOOLATSUEEILZF5NGAGFS3JCUXG4I6EWTMTWVRLTJDGA6DKG5NKPA4",
    "snd": "DF2QZX26LUCBOGLQWAXJT56OJFHSZ2V4UFD3SEXXASEVZAIZV6FZSJ2HPM",
    "type": "pay"
  }
}
```


- single-account signing

Constructing a payment TX using the python SDK

Step 3

- Send the transaction to a node

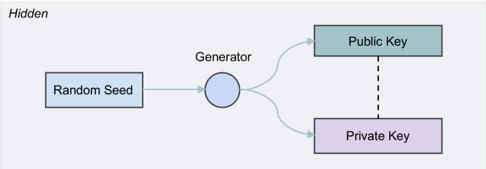
```
txid=algodClient.send_transaction(signedTx)
```

What is missing?

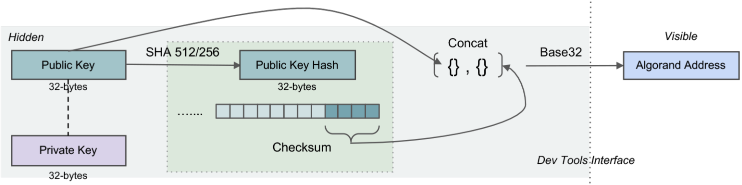
- the sender address: `sAddr`
- the receiver address: `rAddr`
- the sender key: `sKey`
- the node object: `algodClient`

Algorand keys and addresses

- signature algorithm `EdDSA` based on elliptic curve ▸ Ed25519
- the key generation algorithm generates a pair of public (`verification`) and private (`signing`) keys from a random seed.

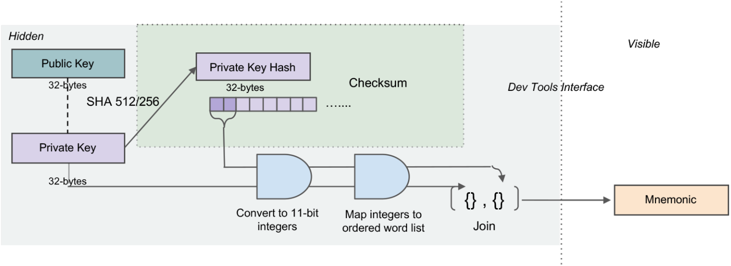


- from a public key (32 bytes) to a human readable `address` (58 bytes)



Algorand keys and addresses

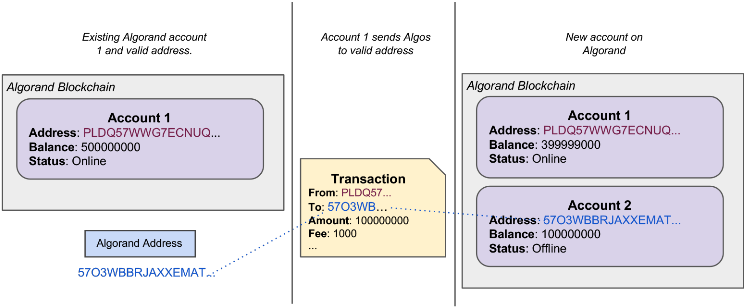
- Private keys are a sequence of 32 bytes and are stored in human readable from using a sequence of 25 English words called the `mnemonic`.



Algorand Accounts

An `address` generated from a `public key` can identify an `account` on the `blockchain`.

- Giving an account a balance



- minimum balance of 100000 microAlgos
- `online` if it participates in consensus (see later)
- `offline` if it does not participate in consensus

Source: https://developer.algorand.org/docs/get-details/accounts/

Generating an account using the python SDK

```
from algosdk import account, mnemonic
private_key,address = account.generate_account()
mnemonic.from_private_key(private_key)
```

Code

Folder 01-SinglePayTx

- `creatSingle.py` creates single signature address
  takes account name on command line and stores address and mnemonic in `.addr` and `.mnem`
- `singlePay.py` creates a payment transaction of 1 Algo
  takes sender mnem and receiver addr plus node directory on command line, performs transaction and writes signed and unsigned transactions in the `TX` folder
- use the following command to see the transactions

```
goal clerk inspect <filename>
```

Algorand Multisignature Accounts

- an ordered set of addresses `Addr` with a threshold `thr` and version
- like a regular address: send transactions and participating in consensus
- to sign a transaction with a multisignature account as a sender, at least `thr` signature from set `Addr` are needed.

Creating a Multisignature Account

- `msig = Multisig(version, threshold, accounts)`
  - `version` : the multisig version
    current value: 1
  - `threshold` : how many signatures are necessary
  - `accounts` : a list of addresses

Constructing a payment TX using the python SDK

the sender is a multisignature account

- Step 1: Create the transaction

```
unsignedTx=PaymentTxn(sAddr,params,rAddr,amount,None,note)
```

same as before but `sAddr` is a multisig address


```
{
  "txn": {
    "amt": 1000000,
    "fee": 1000,
    "fv": 17279990,
    "gen": "testnet-v1.0",
    "gh": "SGO1GKSzyE7IEPItTxCByw9x8FmnrCDexi9/cOUJOiI=",
    "lv": 17280990,
    "note": "Q2lhbyBNdWx0aVBpbm8hISE=",
    "rcv": "J6N2FN6E4M5IPNIGLVVWSLNSQNXGLY4I4RT3R5MALDKHAQS4XGG42DNO7A",
    "snd": "KWTRU5DRUAJNEX3LQKFG5KWQDQNO6INUBGJHZERXHB5SIS3YYHHQWLCL6E",
    "type": "pay"
  }
}
```

Constructing a payment TX using the python SDK

- Step 2: Add the addresses and the threshold

```
mTx=MultisigTransaction(unsignedTx,mSig)
```


```
{
  "msig": {
    "subsig": [
      {
        "pk": "4QPE5LAWA3JH2Q362ZJREY2GT6POVCK2SVWKGUOMUWYIXPK2OQJW3Q7SXQ"
      },
      {
        "pk": "CZZB24IFYYD5LL5RBHPH6EMXHS7EVW66ASN32HZKICWHNCIRBOSHMRXT3U"
      },
      {
        "pk": "J6N2FN6E4M5IPNIGLVVWSLNSQNXGLY4I4RT3R5MALDKHAQS4XGG42DNO7A"
      }
    ],
    "thr": 2,
    "v": 1
  },
  "txn": {
    "amt": 1000000,
    "fee": 1000,
    "fv": 17279990,
    "gen": "testnet-v1.0",
    "gh": "SGO1GKSzyE7IEPItTxCByw9x8FmnrCDexi9/cOUJOiI=",
    "lv": 17280990,
    "note": "Q2lhbyBNdWx0aVBpbm8hISE=",
    "rcv": "J6N2FN6E4M5IPNIGLVVWSLNSQNXGLY4I4RT3R5MALDKHAQS4XGG42DNO7A",
    "snd": "KWTRU5DRUAJNEX3LQKFG5KWQDQNO6INUBGJHZERXHB5SIS3YYHHQWLCL6E",
    "type": "pay"
  }
}
```

Constructing a payment TX using the python SDK

- Step 3: Add `threshold` signatures

`mTx.sign(privateKey)`


```
{
  "msig": {
    "subsig": [
      {
        "pk": "4QPE5LAWA3JH2Q362ZJREY2GT6POVCK2SVWKGUOMUWYIXPK2OQJW3Q7SXQ",
        "s": "ZZDoibOTxPf3OUhp08hMwjEHhe60M9pPRa4ayw5mduxjw7eCmYO+0WoMFig2GG+v+p8c5FpNfGLuKsPO82oTAw=="
      },
      {
        "pk": "CZZB24IFYYD5LL5RBHPH6EMXHS7EVW66ASN32HZKICWHNCIRBOSHMRXT3U",
        "s": "GcrY/ZNHR4f44aefPCcYfouTcbJivjZ1kvCCWFxDU5+HAR+AdvNKEXLhZA/aBwypyHmFnkOt9jy1j+ypP71ZCg=="
      },
      {
        "pk": "J6N2FN6E4M5IPNIGLVVWSLNSQNXGLY4I4RT3R5MALDKHAQS4XGG42DNO7A"
      }
    ],
    "thr": 2,
    "v": 1
  },
  "txn": {
    "amt": 1000000,
    "fee": 1000,
    "fv": 17279990,
    "gen": "testnet-v1.0",
    "gh": "SGO1GKSzyE7IEPItTxCByw9x8FmnrCDexi9/cOUJOiI=",
    "lv": 17280990,
    "note": "Q2lhbyBNdWx0aVBpbm8hISE=",
    "rcv": "J6N2FN6E4M5IPNIGLVVWSLNSQNXGLY4I4RT3R5MALDKHAQS4XGG42DNO7A",
    "snd": "KWTRU5DRUAJNEX3LQKFG5KWQDQNO6INUBGJHZERXHB5SIS3YYHHQWLCL6E",
    "type": "pay"
  }
}
```

Constructing a payment TX using the python SDK

- Step 4: Send the transaction to a node

```
txid=algodClient.send_transaction(mTx)
```

Logic signatures

Logic: Raw Program Bytes (required)
Sig: Signature of Program Bytes (Optional)
Msig: Multi-Signature of Program Bytes (Optional)
Args: Array of Bytes Strings Passed to the Program (Optional)

A `LogicSig` is associated with a TEAL program and it is valid for a transaction

- the hash of the program is equal to the sender account of the transaction
- the `Sig` field contains a valid signature of the program from the sender single account
- the `Msig` field contains a valid multi-signature of the program from the sender multi-signature account

Contract account

- Each TEAL program is associated to a unique Algorand address
  - the address is a hash
  - use `goal clerk compile` to get it
- a *contract account* looks like any other Algorand account
  - can send token and assets to it
- a spending transaction from a contract account must provide arguments to the TEAL program that make the program *happy*

Delegated approval

- An account (single or multi) can sign a TEAL program
- Anyone can spend from the account according to the logic of TEAL
  - a CEO can delegate an assistant to withdraw from a corporate account to his personal account up to a certain limit
  - the TEAL program checks the limit and the destination and approves the transaction
  - the CEO signs the program
  - the assistant sends the transaction to the blockchain everytime a withdraw is needed

Contract account

A transaction can be signed by a TEAL program:
Transaction Execution and Approval Language

- a TEAL **program**
- **arguments** to the TEAL program

such that

- the TEAL program on the arguments provided terminates the execution with **one single non-zero element** in the stack
- the hash of the TEAL program is equal to the **address of the sender** of the transaction

Source: https://developer.algorand.org/docs/get-details/dapps/smart-contracts/frontend/smartsigs/

Signing with a program

- Write the TEAL program
- Load the Program Bytes into the SDK.
- Create a Logic Signature based on the program.
- Create the Transaction.
- Set the `sender` of the transaction to the TEAL program address
- Sign the Transaction with the Logic Signature.
- Send the Transaction to the network.

A simple TEAL program

```
byte "34"
arg 0
==
txn CloseRemainderTo
txn Receiver
==
&&
```

It runs with success iff

- The first argument is "34"
- The `rcv` address and the `close` address are the same

Signing a transaction with a TEAL program

Step 1: create the logic

```
# Read TEAL program
data=open(myprogram, 'r').read()

# Compile TEAL program
response=algodClient.compile(data)
print("Response Hash = ", response['hash'])

# Create logic sig
programstr=response['result']
t=programstr.encode()
program=base64.decodebytes(t)

arg_str = "34"
arg1=arg_str.encode()
lsig=transaction.LogicSig(program, args=[arg1])
```

Signing a transaction with a TEAL program

Step 2: create the transaction

```
sender="J7KCCCMV2EE3KZS2DH7FFFMHGC4YLSAQVRSC4CMYPSP3IG62YD44MFAM04"
receiver="IOYH24KGB6FMWWXDN3TUW35G6TUK45IQEOVKZKR6Z6MOLXURODKFCXX64Q"
closeremainderto=receiver
params=algodClient.suggested_params()
txn = transaction.PaymentTxn(
    sender, params, receiver, 30000, closeremainderto)
```

The sender address is the hash of the TEAL program as output by `algodClient.compile`

Signing a transaction with a TEAL program

Step 3: add the logic to the transaction

```
# Create the LogicSigTransaction with contract account LogicSig
lstx = transaction.LogicSigTransaction(txn, lsig)
transaction.write_to_file([lstx], "simple.stxn")
# Send raw LogicSigTransaction to network
txid = algodClient.send_transaction(lstx)
```

A transaction signed by a Program


```
{
  "lsig": {
    "arg": [
      "MzU="
    ],
    "l": "#pragma version 1\nbytecblock 0x3334\nbytec_0 // \"34\"\narg_0\n==\ntxn CloseRemainderTo\ntxn Receiv
\n==\n&&\n"
  },
  "txn": {
    "amt": 30000,
    "close": "IOYH24KGB6FMWWXDN3TUW35G6TUK45IQEOVKZKR6Z6MOLXURODKFCXX64Q",
    "fee": 1000,
    "fv": 17361459,
    "gen": "testnet-v1.0",
    "gh": "SGO1GKSzyE7IEPItTxCByw9x8FmnrCDexi9/cOUJOiI=",
    "lv": 17362459,
    "note": "jnFqu7AbJLE=",
    "rcv": "IOYH24KGB6FMWWXDN3TUW35G6TUK45IQEOVKZKR6Z6MOLXURODKFCXX64Q",
    "snd": "J7KCCCMV2EE3KZS2DH7FFFMHGC4YLSAQVRSC4CMYPSP3IG62YD44MFAMO4",
    "type": "pay"
  }
}
```


The hash of the field `lsig.l` is equal to the `sender` field

Delegated Signatures

Smart Signatures

- Write the TEAL program
- Load the Program Bytes into the SDK.
- Create a Logic Signature
- Create the Transaction.
- Set the `sender` of the transaction to the TEAL program address
- Sign the Transaction with the Logic Signature.
- Send the Transaction to the network.

Delegated Signatures

Delegated Signatures

- Write the TEAL program
- Load the Program Bytes into the SDK.
- Create a Logic Signature
- Sign the Logic Signature with a specific account.
- Create the Transaction.
- Set the `sender` of the transaction to the address that signed the logic.
- Sign the Transaction with the Logic Signature.
- Send the Transaction to the network.

How to use the delegated signature

- The sender S creates a logic, signs it and gives it to the delegator D.
- The program checks the conditions under which algos will leave the sender's account.
- The delegator D can create a transaction to receiver Algos from S's account without S's intervention.
- The payment transaction is signed by D by using the signed logic received from S

Step 1 (modified): create the logic and sign it

```
# Read TEAL program
data = open(myprogram, 'r').read()

# Compile TEAL program
response=algodClient.compile(data)
programAddr=response['hash']
programstr=response['result']
print("Response Result = ",programstr)
print("Response Hash   = ",programAddr)

# Create logic sig
t = programstr.encode()
program = base64.decodebytes(t)

# Create arg to pass if TEAL program requires an arg,
# if not, omit args param
# string parameter

arg_str = "34"
arg1=arg_str.encode()
lsig=transaction.LogicSig(program, args=[arg1])

passphrase=open(senderMNEMFile,'r').read()
senderKey=mnemonic.to_private_key(passphrase)
senderAddr=account.address_from_private_key(senderKey)
print("Address of Sender/Delegator: " + senderAddr)
lsig.sign(senderKey)
```

Algorand Standard Assets (ASAs)

- Coins living on top of the Algorand blockchain
- Many instance of the same type. AKA *fungible assets*

Parameters:

- Creator (required)
- AssetName (optional, but recommended)
- UnitName (optional, but recommended)
- Total (required)
- Decimals (required)
- DefaultFrozen (required)
- URL (optional)
- MetaDataHash (optional)

Source https://developer.algorand.org/docs/get-details/asa/

Algorand Standard Assets (ASAs)

Four different types of users associated with an ASA

- **Manager**: can re-configure or destroy an asset
- **Reserve**: all non-minted assets will reside in this account. Purely informational.
- **Freeze**: can freeze and unfreeze the asset for a specific account
- **Clawback**: account that is allowed to transfer assets from and to any asset holder

ASA Creation

Restrictions

- A single account can create up to 1000 assets
- For each asset the account creates or owns the minimum balance is increased by .1 Algos
- An account can receive only assets for which it has opted-in

ASA Creation – Python SDK

```
txn=AssetConfigTxn(
    sender=creatorAddr,
    sp=params,
    total=1000,
    default_frozen=False,
    unit_name="OctoT",
    asset_name="OctoAsset",
    manager=managerAddr,
    reserve=reserveAddr,
    freeze=freezeAddr,
    clawback=clawbackAddr,
    url="https://giuper.github.io",
    decimals=0)
```

Then the transaction is signed and sent to a client.

ASA Creation – TX


```json
{
  "txn": {
    "apar": {
      "am": "gXHjtDdtVpY7IKwJYsJWdCSrnUyRsX4jr3ihzQ2U9CQ=",
      "an": "My New Coin",
      "au": "developer.algorand.org",
      "c": "EW64GC6F24M7NDSC5R3ES4YUVE3ZXXNMARJHDCCCLIHZU6TBEOC7XRSBG4",
      "dc": 2,
      "f": "EW64GC6F24M7NDSC5R3ES4YUVE3ZXXNMARJHDCCCLIHZU6TBEOC7XRSBG4",
      "m": "EW64GC6F24M7NDSC5R3ES4YUVE3ZXXNMARJHDCCCLIHZU6TBEOC7XRSBG4",
      "r": "EW64GC6F24M7NDSC5R3ES4YUVE3ZXXNMARJHDCCCLIHZU6TBEOC7XRSBG4",
      "t": 50000000,
      "un": "MNC"
    },
    "fee": 1000,
    "fv": 6000000,
    "gh": "SGO1GKSzyE7IEPItTxCByw9x8FmnrCDexi9/cOUJOiI=",
    "lv": 6001000,
    "snd": "EW64GC6F24M7NDSC5R3ES4YUVE3ZXXNMARJHDCCCLIHZU6TBEOC7XRSBG4",
    "type": "acfg"
  }
}
```


Type `acfg` stands for *asset configuration*

The missing `caid` and the presence of `apar` field distinguish the TX as an asset *creation* transaction

Opting in an ASA – Python SDK

- Before an account can receive a specific asset it must opt-in to receive it.
- An opt-in transaction places an asset holding of 0 into the account and increases its minimum balance by 100,000 microAlgos.
- An opt-in transaction is simply an asset transfer with an amount of 0, both to and from the account opting in.

```
txn=AssetTransferTxn(sender=holderAddr,
        sp=params,receiver=holderAddr,amt=0,index=assetID)
```

Opting in an ASA – TX

```
{
  "txn": {
    "arcv": "QC7XT7QU7X6IHNRJZBR67RBMKCAPH67PCSX4LYH4QKVSQ7DQZ32PG5HSVQ",
    "fee": 1000,
    "fv": 6631154,
    "gh": "SGO1GKSzyE7IEPItTxCByw9x8FmnrCDexi9/cOUJOiI=",
    "lv": 6632154,
    "snd": "QC7XT7QU7X6IHNRJZBR67RBMKCAPH67PCSX4LYH4QKVSQ7DQZ32PG5HSVQ",
    "type": "axfer",
    "xaid": 168103
  }
}
```

- The `axfer` distinguishes this as an asset transfer transaction with asset id specified by `xaid`
- an optin TX:
  - Same address for sender and receiver
  - No amount specified

Transfer of an ASA

```
txn=AssetTransferTxn(sender=senderAddr,sp=params,
          receiver=receiverAddr,amt=10,index=assetID)
```


```json
{
  "txn": {
    "aamt": 1000000,
    "arcv": "QC7XT7QU7X6IHNRJZBR67RBMKCAPH67PCSX4LYH4QKVSQ7DQZ32PG5HSVQ",
    "fee": 3000,
    "fv": 7631196,
    "gh": "SGO1GKSzyE7IEPItTxCByw9x8FmnrCDexi9/cOUJOiI=",
    "lv": 7632196,
    "snd": "EW64GC6F24M7NDSC5R3ES4YUVE3ZXXNMARJHDCCCLIHZU6TBEOC7XRSBG4",
    "type": "axfer",
    "xaid": 168103
  }
}
```


The transaction must be signed by the sender.

Revoking asset

- Revoking an asset for an account removes a specific number of the asset from the revoke target account.
- Revoking an asset from an account requires specifying sender (the revoke target account) and an asset receiver (the account to transfer the funds back to).
- It must be signed by the `clawback` address
- `AssetTransferTxn`
  - `sender` (the clawback address)
  - `sp` (the parameters)
  - `receiver` (the receiving address)
  - `amt` (amount of assets)
  - `index`
  - `revocation_target` (the address that will lose the asset)

Revoking an ASA – TX


```
{
  "txn": {
    "aamt": 500000,
    "arcv": "EW64GC6F24M7NDSC5R3ES4YUVE3ZXXNMARJHDCCCLIHZU6TBEOC7XRSBG4",
    "asnd": "QC7XT7QU7X6IHNRJZBR67RBMKCAPH67PCSX4LYH4QKVSQ7DQZ32PG5HSVQ",
    "fee": 1000,
    "fv": 7687457,
    "gh": "SGO1GKSzyE7IEPItTxCByw9x8FmnrCDexi9/cOUJOiI=",
    "lv": 7688457,
    "snd": "EW64GC6F24M7NDSC5R3ES4YUVE3ZXXNMARJHDCCCLIHZU6TBEOC7XRSBG4",
    "type": "axfer",
    "xaid": 168103
  }
}
```


- type is asset transfer `axfer`
- `asnd` is the address from which the assets will be revoked and identifies this as a revoke transaction
- `snd` is the clawback address

Other operations

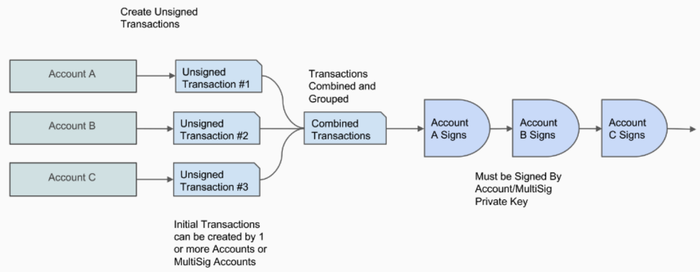
- The manager, freeze, clawback, reserve addresses can be modified.
  - Must be signed by the `Manager` address
  - An asset creation transaction with a specified asset id
- An asset can be frozen/unfrozen with an `AssetFreezeTxn`.
  - Must be signed by the `Freeze` address
  - `new_freeze_state=True/False`
- The asset held by an account can be revoked by the `Clawback`
- An asset can be destroyed by the `Manager`.
  - The creator must hold all the assets.
  - Asset configuration transaction with asset ID (unlike the creation transaction) and no asset parameter (unlike a reconfiguration)

Atomic Transfers

- atomic transfers are implemented as irreducible batch operations
  - a group of transactions are submitted as a unit
  - all transactions in the batch either pass or fail.
- eliminates the need for more complex solutions like hashed timelock contracts that are implemented on other blockchains.

Source https://developer.algorand.org/docs/get-details/atomic_transfers/

The flow of an atomic transfer



Atomic Transfer Flow

Step by step

- Each user creates an unsigned transaction
- The group is created

```
# get group id and assign it to transactions
gid = transaction.calculate_group_id([txn_1, txn_2])
txn_1.group = gid
txn_2.group = gid
```

- Each user signs his own transaction.
  - The field `group` will guarantee that the transaction, even if it is signed, cannot be submitted by itself and will be considered valid only if submitted jointly with the other transactions of the group.
- The list of signed transactions is submitted to the blockchain.

Distributed Applications

Smart contracts (or distributed applications or dApps) have a state

- global variables (i.e., all addresses see the same value)
- local (i.e., per address) variables.

Life Cycle of a dApp

- Write the dApp programs:
  - write in TEAL and then compile into AVM bytecode
  - write in PyTEAL, compile to get TEAL and then as above
  - ...
- Create the dApp by deploying it on the blockchain
  - an index, to be specified in all further calls
  - an address, to be specified to send algos and assets to the dApp
- Addresses that wish to use the dApp must optin

```
utxn=ApplicationOptInTxn(Addr,params,index)
```

- Once you have opted in, you can call the dApp

```
utxn=ApplicationNoOpTxn(Addr,params,index)
```

```
appArgs=[stringVar.encode(),intVar.to_bytes(8,'big')]
utxn=ApplicationNoOpTxn(Addr,params,index,appArgs)
```

Creating a dApp I

```
utxn=ApplicationCreateTxn(creatorAddr,params,on_complete, \
                          approvalProgram,clearProgram, \
                          globalSchema,localSchema)
```

- `creatorAddr`: address of the creator
- `params`: parameters of the transaction
- `approvalProgram`: code to be executed when
  - `NoOp` generic execution call of the dApp.
  - `OptIn` an address decides to participate to the dApp and its local storage is enabled.
  - `DeleteApplication` when the dApp is removed
  - `UpdateApplication` when the dApp TEAL program is updated
  - `CloseOut` close the address participation in the dApp without removing it from the address balance.
- clearProgram: code to be executed when an address wants to remove the dApp from its balance account
- `globalScheme` and `localScheme` specify the number of integer and string global and local variables

Creating a dApp II

The `approval` and `clear` programs must be compiled:

- inline

```
clearProgramSource=b"""#pragma version 4 int 1 """
clearProgramResponse=algodClient.compile(clearProgramSource.decode('utf-8'))
clearProgram=base64.b64decode(clearProgramResponse['result'])
```

- read from a file

```
with open(approvalFile,'r') as f:
    approvalProgramSource=f.read()
approvalProgramResponse=algodClient.compile(approvalProgramSource)
approvalProgram=base64.b64decode(approvalProgramResponse['result'])
```

Creating a dApp III

Constructing `global` and `local schema`

```
global_ints=2
global_bytes=1
globalSchema=StateSchema(global_ints,global_bytes)
```

```
local_ints=2
local_bytes=1
localSchema=StateSchema(local_ints,local_bytes)
```